Visual User Interface API

for eMESA Software

Field Session 2009, Colorado School of Mines

Billy Leavenworth

Mila Rodriguez

Doug Welch

Table of Contents

Table of Figures

1. Abstract

Mining and construction companies often use databases to create and manage work orders, but the large scale of these databases makes it an inefficient, time-consuming process. Dimensional Technology Solutions (DTS) sells a user interface known as the **e**lectronic **M**aintenance **E**xecution **S**cheduling **A**ccelerator (eMESA) that streamlines the process of managing work orders and scheduling. DTS would like a browser-based touch screen interface that will make eMESA quick and intuitive for use with Smartboards and other touch screens.

Using the internet multimedia language Silverlight, our team created a visual application programming interface (API) for the visual eMESA user interface. The API includes support for a variety of behaviors and functionalities, including the nesting, movement, merging, and resizing of multiple objects. When interfaced with the eMESA framework, this API allows for the creation of extremely powerful interfaces with practically endless possibilities in terms of future expansion.

2. Introduction

Dimension Technology Services (DTS) is a software development company whose products feature support services for asset intensive industries. Their flagship product is a piece of software called eMESA, which provides companies with a single interface through which to perform a variety of tasks including planning, scheduling, and work order execution. eMESA in its current form is a web-based application that is convenient for upper-level management. An intuitive, purely visual interface that could be used on a touch screen system would provide an extremely user-friendly way for users of all management levels to quickly and effectively assign tasks and allocate resources.

The visual user interface API needed to provide a way to dynamically represent resources, such as personnel and equipment, and allow them to be reassigned or re-associated with other resources and work areas with the touch of a finger. With a quick tap, the user would be able to learn more about a given resource, scheduled task, or operating efficiency. A hierarchical zoom function would allow allow a foreman to supervise assignments in his location, a shift manager to assign work orders and shift personnel between locations and assignments, and upper management to view all activity pertaining to their work sites. Layouts and resource arrangements could be saved and reloaded from future use. All of this information would be hosted on a central server and relayed in real time.

3. Requirements

The client specified a number of functional and non-functional requirements for the project. Functional requirements define a function of the software, while non-functional requirements define other non-software related criteria, such as hardware requirements.

3.1. Functional Requirements

- The software must interface with the existing eMESA system.
- Elements representing resources, locations, and work forms can be created, moved, and deleted
- Some database objects must act as containers, so that the appropriate records in the database can be dragged in and out and assigned relationships.
- The system should implement "drag and drop" functionality.
- The interface must recognize user roles, e.g. foreman or mechanic, and enforce privileges accordingly.
- Users should be able to zoom in on objects to see additional details.

3.2. Non-Functional Requirements

- The interface should run in a web browser (Internet Explorer or Firefox).
- The interface should be made using CSS/HTML/JavaScript *or* Silverlight/C#.

3.3. Scope

The initial scope of the project called for the creation of a functional visual interface for the eMESA system. Upon later consideration of time constraints, technological limitations, and the complexity of the eMESA system as a whole, the scope was reduced to the creation of an API for the later development of a fully functional visual interface. The refactored scope did not call for any interaction between the API and the eMESA servers, leaving that aspect of the requirements for future development.

The following are the requirements as determined by the initial scope of the project.

Requirements:

- Connecting with eMESA.
- Mapping a database object as a container or element.
- Basic touch screen features and functionality:
    - Create/resize/destroy items
    - Moving items around the screen.
    - Zooming in/out and scrolling on the screen.

Optional desired functionality:

- Providing support for functional limitations based on user roles.
- Saving to a template.

3.4. User Stories

The client provided a number of user stories to illustrate how a user would interact with the desired features of the system. These stories are as follows.

1. A user can create a new container
    - the container should initially be placed in a general work space
    - the container should not overlap with other containers
2. A user can drag a container and drop it on the work space, where it will snap to a column-

based layout.

- the container will "bump" existing containers in the target column
- one container can be dragged into another container
- a container can be placed between two containers

3. A user can define the properties of a container (the name, color, etc.)
4. The user can zoom in and out of the center of the current view.
5. The user can create a new column in the work space layout.

4. Design

4.1 Technologies

The development of the API involved a number of different technologies and programming languages for various tasks. These technologies included: Silverlight, XAML, C#, and WCF.

Silverlight

Microsoft Silverlight is a web-browser plugin that provides support for media-rich internet applications that include such features as animation, vector graphics, audio-video playback, and .NET language feature support. [1] As a web-browser plugin, Silverlight provides functionality across a large variety of web-browsers and operating systems. [2]

Silverlight consists of two layers of code: a visual layer defined by XAML, and a dynamic layer defied in C#. Both layers considered as a whole define a Silverlight *user control*.

XAML

XAML, or Extensible Application Markup Language, is an XML-based language used by Silverlight for initializing visual elements. Its markup is very similar to other web scripting languages such as XHTML and CSS. [3]

C#

C# is an object-oriented programming language developed by Microsoft that is used by Silverlight to provide dynamic functionality and to enable inter- and intra-XAML interaction. Syntactically C# is extremely similar to Java and, to a lesser extent, C++. As a .Net language, C# is usually restricted to Windows-based operating systems, though this limitation can be overcome via the use of Silverlight. [4]

WCF

WCF, or *Windows Communication Framework*, is a .NET framework that provides the ability for a wide range of internet services to communicate with each other. [5] Simply put, it serves as a translator for internet-based applications running different languages, therefore allowing the applications to communicate with each other. As a part of the .NET framework, WCF is coded in a subset of C# with extra mark-up.

4.2 High Level Design

The Silverlight User Interface is a web application specially designed for use with a touch screen. The Touchscreen displays the application via a web browser, from wherein the user can interact with the various components of the interface. The Silverlight application itself communicates with the eMESA Live™ system using WCF protocols to exchange information (see **Figure 1**). The WCF layer allows for the communication of two very different technologies, eMESA and Silverlight, and provides an additional layer of security for their interaction. For technical and security reasons, the Silverlight application does not directly access the information within eMESA databases, but rather sends commands through WCF asking the eMESA servers to do various tasks, return certain information, and so forth. eMESA interacts with the company's SAP business management software and the database to retrieve and process information.

When a user interacts with the touch screen and changes information within the Silverlight application, a WCF call is passed to eMESA, asking the eMESA server to affect changes to the database based on the user's input. Depending on a number of factors, including the user's credentials and the properties of the resources being affected, eMESA will attempt to affect these changes.

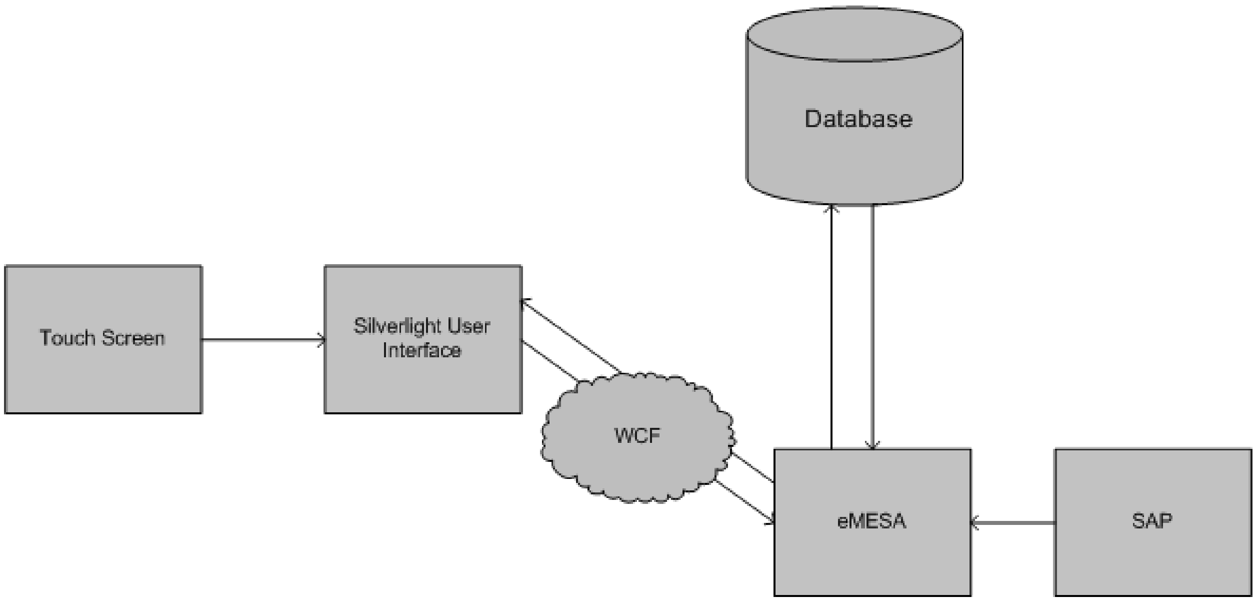
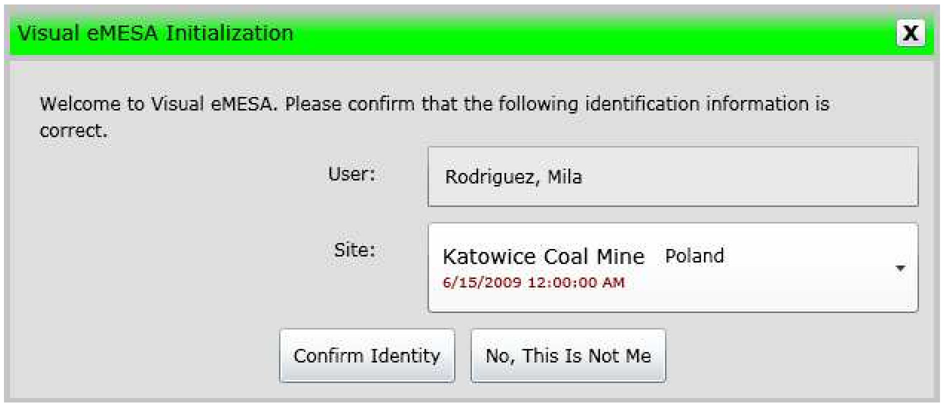


Figure 1: High Level Design

4.3 Low Level Design

When the application is first launched, a Silverlight user control called NewUserPopup retrieves user information from the eMESA web server using the WCF framework (see **Figure 2, 3**). The information retrieved from the server includes the current user's name, role, and the work site being administered. The calls to the WCF framework are administered by a C# class called EMESAInterfacer which sends specially worded queries to a service reference included on the eMESA server. The exact methods and functionality of EMESAInterfacer are not shown on the attached UML diagram as they include proprietary information.



Welcome to Visual eMESA. Please confirm that the following identification information is correct.

User: Rodriguez, Mila

Site: Katowice Coal Mine Poland 6/15/2009 12:00:00 AM

Figure 4: The NewUserPopup control retrieves information from the eMESA web server.

The actual Silverlight application is managed by a user control called Page, which provides a canvas on which the different custom controls interact. The user controls can be organized into two basic subsystems based on common functionality: managerial classes and visual components. All elements visible to the user must exist on the Page, though they may be controlled dynamically from other classes.

The API includes three basic types of visual components: popups, elements, and containers. Popups are extensions of the built-in Silverlight popup control which prompt the user for basic information in order to perform a given task. Elements are the most basic of the visual components, and are designed to designate components which cannot have items nested within them (see **Figure 5**). The final visual components, containers, can have elements or other containers nested within themselves (see **Figure 6**).

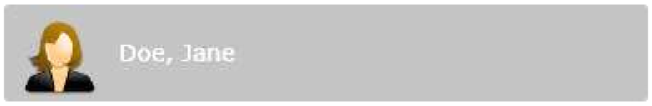


Figure 5: The element component is the most basic visual component.

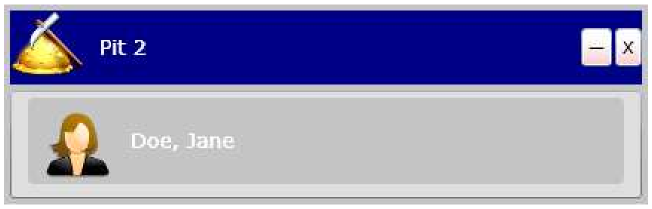


Figure 6: A container component with a nested element.

A variety of managerial classes control the behavior of the visual components (see **Figure 3**). The managerial classes have no visual representation, and are written purely in C#. Each manager controls exactly one type of behavior, with little to no overlap between behaviors. The API includes four managerial classes: DragDropManager, IPhoneScrollManager, PopupManager, and TopContainerResizeManager.

DragDropManager allows containers and elements to be dragged around the screen and dropped into a grid or into other containers.

IPhoneScrollManager provides a panning/click-to-scroll behavior that is similar to the scrolling functionality on the Apple iPhone.

PopupManager controls the display of the various popups and provides a way for other components to obtain and interact with the information inputted by the user.

TopContainerResizeManager allows containers to be resized by clicking on the border and dragging to a new position. This functionality is similar to the drag-to-resize behavior of windows in the Microsoft Windows operating system.

In addition to these managers, the managerial classes also include two interfaces: Mask and Draggable. Mask provides an additional layer of security for eMESA interaction, providing a C#

object for the Silverlight application to modify so that changes do not directly affect an eMESA objects. Different masks exist for each the type of resource that exists on eMESA, though the specific functionalities and properties of the masks contain sensitive proprietary information and will not be discussed here.

The Draggable interface defines objects that can be dragged, dropped, and nested within other objects by the various managers. By using this interface, the amount of code needed to apply this functionality to both elements and containers was effectively halved.

The remaining classes, GhostColumn and Shadow, are graphical prompts used by the DragDropManager. Shadow draws an outline in the workspace to represent the possible drop locations based on cursor position. GhostColumn allows the use to drop a component into a new column that is represented as a shaded area on the workspace.

5. Implementation and Results

5.1. Design Decisions on Language

The client gave us the choice of using either Silverlight or a combination of HTML/CSS/JavaScript to use on the project. There were our only options in terms of language due to the fact that the final product needed to work with either Firefox or Internet Explorer, and could not be limited to a single operating system.

We chose to use Silverlight for a variety of reasons. Silverlight could easily interact with the eMESA system, which is written in C# on a .NET framework. Additionally, Silverlight has functionality that parallels Flash and generates a configurable web page that can be run from most browsers. As the application is run within a web browser, it will work with any touch screen with an internet connection.

Another consideration we had was the fact that some team members had experience working with Visual C++ and the .NET framework in previous courses. At the same time while all team members had some familiarity with the HTML/CSS/JavaScript technologies, none were comfortable with their level of knowledge to attempt developing a product of this scope in those languages.

5.2. Design Decisions on Development and Recycled Code

Silverlight technology is extremely new. Silverlight 1.0 was initially released in April 2007, making the software currently just over two years old. As a result, there is some functionality that simply doesn't exist, some user controls are poorly developed, documentation is lacking, and seemingly simple tasks take many lines of code to execute.

As Silverlight is intended to be a lightweight browser plugin, it does not support the entire C# library, but rather a small subset of C# that includes the most common libraries, elements, and methods used by programmers. Unfortunately, this means that some more advanced elements are not included within this subset due mostly to the fact that they are used more often by more experienced programmers rather than beginners, and therefore are not considered "common" enough for the Silverlight C#. One such example is the HashSet, a data structure which is not supported by Silverlight.

Additionally, some functionality is simply missing, probably because it was simply not a priority in the first two releases of Silverlight. On the other hand, sometimes the functionality does exist, but the amount of code involved to achieve it is mind-boggling. Silverlight supports a XAML object called ScrollViewer, which is simply a panel that automatically provides scrollbars once its content exceeds specified dimensions. By default, the ScrollViewer has a thin, grey border. In most other XAML objects with borders, to remove the border, one simply needs to set the BorderBrush property to Transparent. With ScrollViewer, however, removing the border involves sixty lines of code in which the entire style layout of the viewer must be redefined, and a single variable changed from the default.

Documentation of Silverlight functionality and troubleshooting is unevenly developed. Development tools have little documentation, and custom user controls created by other users seldom work for more than the single purpose they were developed for. While we looked for information on all of the challenges we faced, only some were well documented. The well documented parts included the server-side communications provided by the Windows Communication Foundation (WCF), using a transform to provide centered zooming, adding pop-up windows, and toggling full screen mode.

The biggest challenge lay in designing functionality that simply didn't exist within Silverlight. Many aspects involving dragging and dropping, resizing, minimizing, and scrolling were of our own design.

The drag & drop manager started from an example provided by the developers of SilverlightDesktop.net. Though the example code would not compile, we used it as a starting point to develop our own drag & drop manager. One of the key things about the drag & drop manager is that it uses an overlaying canvas where items are transferred to when they get picked up, rather than simply changing the z-index when moving something. This feature allows the program to keep track of the moving item's original position, which would not be stored otherwise, to allow it to snap back into position should it require. Some design concepts were based on the original example, while much of the rest was entirely original. Original concepts included the ability to only drag an item by its title bar, drawing an outline where something could be placed, and nesting draggable things inside of one another.

An Interface extended the use of drag & drop manager to work with both containers and elements. Allowing panels to manage their own contents rather than using an array also cut back on code (see detail in design issues below). Creating data structures and UIElements dynamically reduced the compile time and overall size of the application.

5.3. Design Issues

The general lack of functionality of the Silverlight language in its current state of development led to what we called a "workaround mentality." Constantly faced with situations where the functionality was only partially implemented, or only worked under certain circumstances, we became so accustomed to thinking up alternative ways to do the same task that should some method not work, we generally did not spend overly long agonizing about how to make it work before switching over to another completely different method that ultimately accomplished the same task.

As a result of this workaround mentality, we rarely had any lingering design issues, because code that didn't work was quickly scrapped for code that did. There were, however, a few design issues that lingered, usually because they really only had a single solution. These more serious design issues are listed here.

Protected 'Hit Test' Function

One issue that seriously affected the design of the drag-drop manager was that the hit test functions were protected. A hit test looks for everything that a point or area may be overlapping with visually. Manual testing ignored nested elements, and elements out of view due to scrollbars. As a result of the accessibility level of the hit test functions, drag and drop functionality could not be fully segregated from the Page work area, and "collision handling" could not be handled within the DragDropManager class. As a result, some parts of DragDropManager had to be exposed publicly, and extra safeguards put in place for security.

Dynamically Changing a Typedef

The drag & drop manager class was designed to work with containers, as determined by a typedef. Another object that needed to be controlled by this class used a different data type, but changing the typedef dynamically was not possible and posed a challenge. As a result, the drag & drop manager had to be redeveloped using an interface, Draggable, to represent both types of movable components.

Correlating Visual Placement With Array Arrangement

Managing items within different stack panels or grids and keeping their locations in order with their visual assortment became more of an issue over time. Originally we used a horizontal stack panel filled with single column grids to align containers visually. These one column grids were also placed into an array, wherein they could be managed explicitly. Once a grid was placed in this array it would be added to the vertical stack panel. After stumbling over this approach enough times we decided to let the design structures in Silverlight manage themselves. Rather than trying to model and predict how the stack panel would behave when things were added to it, we decided to just use the collection of the stack panel's contents whenever we needed something. This significantly reduced the amount of overhead and headaches.

5.4. Results

The final version of the Silverlight visual interface for eMESA achieved all of the required functionality, including:

- Connects to and interacts with the eMESA system.
- Elements representing resources, locations, and work forms can be created, moved, and deleted.
- Certain types of database objects act like containers, and other elements and containers can be dragged in or out.
- Resources can be "dragged and dropped" throughout the workspace.
- Users can zoom in on objects to see them in greater detail.

In addition, some of the optional desirable functional requirements were also incorporated, either completely or partially. These requirements include:

- A secure login screen is provided which can be extended to check for user roles.
- Elements are linked to icons which can be pulled from eMESA, and can be expanded to gather other information via Mask objects.
- A demo template has been hard coded into an alternate release of the program for demonstration purposes to show some possible future uses of the API.

6. Conclusion

The visual user interface API for eMESA software is a Silverlight-based application programming interface that provides basic functionality for the development of visual applications for use on a touch screen. The API includes visual components for popups, childless elements, and containers which can nest other containers and children. In addition, managerial classes add "drag and drop", resizing, scrolling, and zooming functionality, as well as a framework for future interfacing with the eMESA software.

The functionality currently developed is sufficient for the most basic eMESA tasks: assigning personnel to equipment to locations and combinations thereof. More advanced tasks will the development of additional components and modules. Scheduling functionality would require a series of prompts for the user to input information about a scheduled task, as well as corresponding WCF protocols to handle information transfer. Clicking or tapping once on a given component could display information about the selected resource using information stored in that component's mask. Additional functionality would include user roles and limitations of options therein, switching views between day shift and night shift, loading saved user or site information from an external XML file, and so forth.

7. References

[1] MSDN. "Silverlight Architecture." MSDN Silverlight Developer Center. Available at: http://msdn.microsoft.com/en-us/library/bb404713(VS.95).aspx. Retrieved on 15 June 2009.

[2] Microsoft. "Silverlight Installation: System Requirements." Microsoft Silverlight.

Available at: http://www.microsoft.com/silverlight/resources/install.aspx?reason=unsupportedplatform#sysreq. Retrieved on 17 June 2009.

[3] MSDN. "XAML Syntax Terminology." MSDN .Net Framework Developer Center. Available at: http://msdn.microsoft.com/en-us/library/ms788723.aspx. Retrieved on 15 June 2009.

[4] MSDN. "Visual C#." MSDN Visual C# Developer Center. Available at: http://msdn.microsoft.com/en-us/library/kx37x362.aspx. Retrieved on 15 June 2009.

[5] MSDN. ".Net Framework Conceptual Overview." MSDN Visual Studio Development Center. Available at: http://msdn.microsoft.com/en-us/library/zw4w595w.aspx. Retrieved on 15 June 2009.

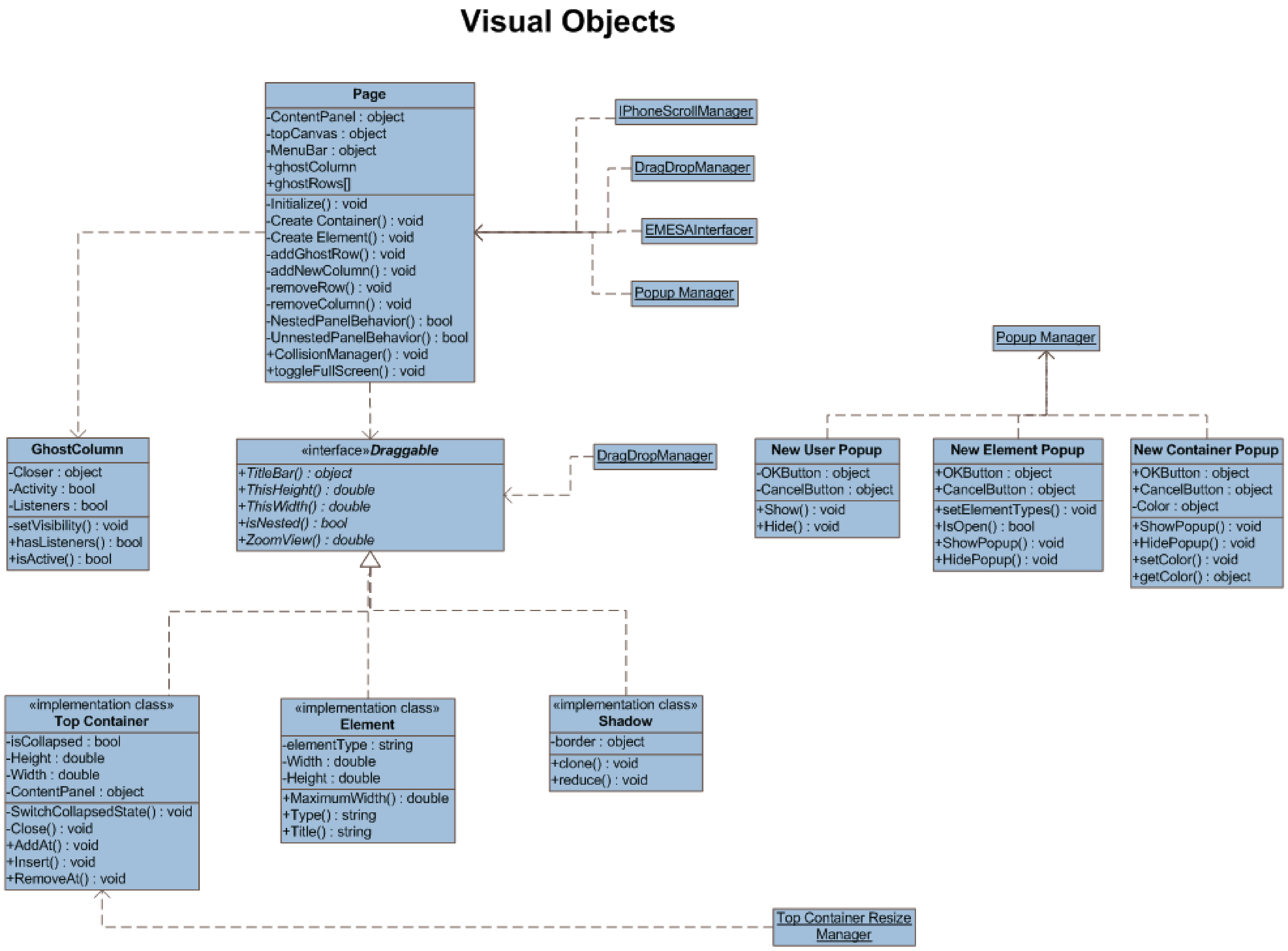


Figure 2: UML Diagram (1)

Managers

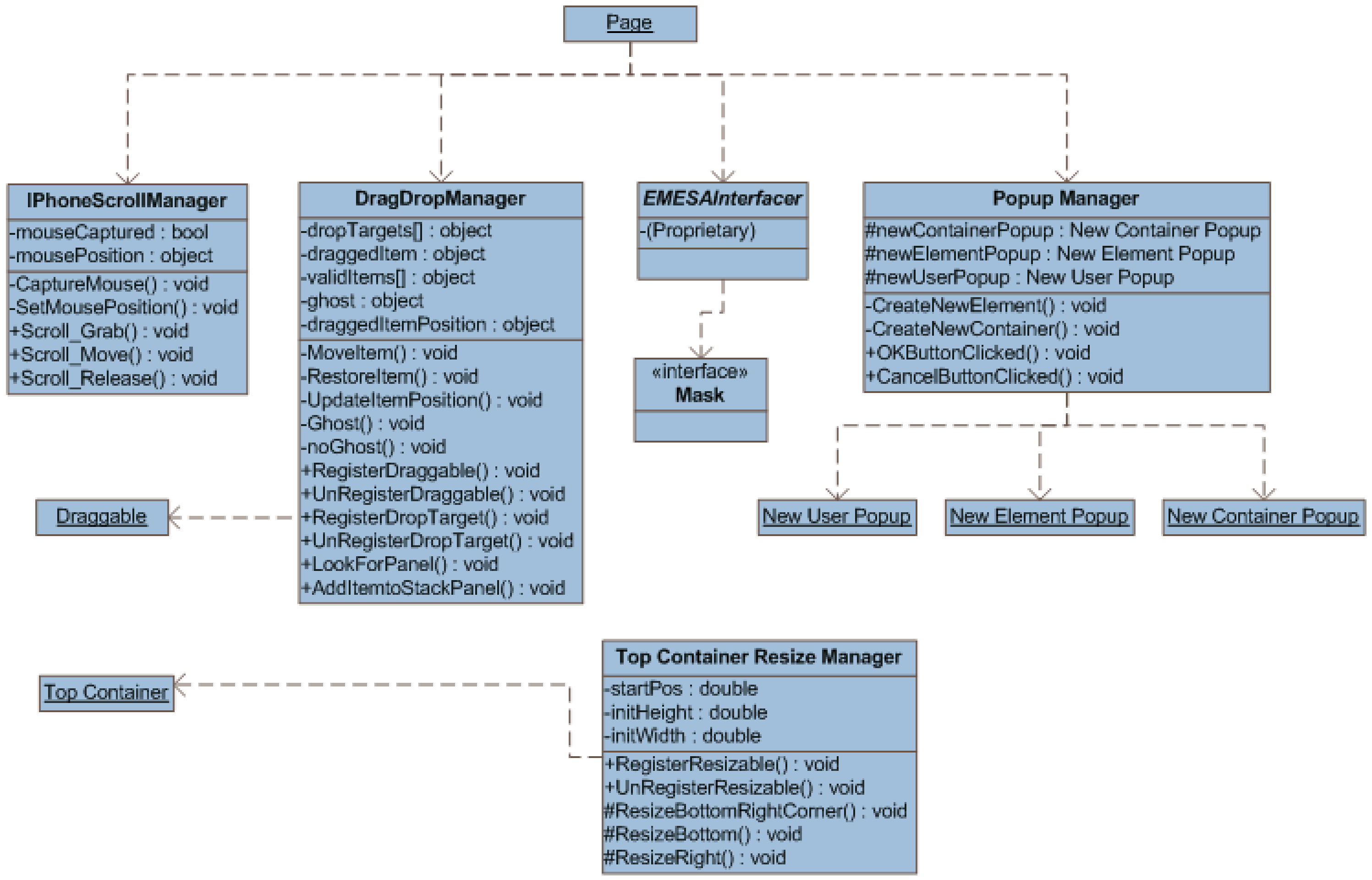


Figure 3: UML Diagram (2)